

Section Solutions 5

Based on handouts by Jerry Cain, Julie Zelenski, and Eric Roberts

Problem One: Double-Ended Queues

```
class Deque {
public:
    Deque();
    ~Deque();

    /* Adds a value to the front or the back of the deque. */
    void pushFront(int value);
    void pushBack(int value);

    /* Returns and removes the first or last element of the deque. */
    int popFront();
    int popBack();

private:
    struct Cell {
        int value;
        Cell* next;
        Cell* prev;
    };
    Cell* head;
    Cell* tail;
};

/* Initially, there are no elements at all. */
Deque::Deque() {
    head = tail = NULL;
}

/* Standard linked-list deletion code. */
Deque::~Deque() {
    while (head != NULL) {
        Cell* next = head->next;
        delete head;
        head = next;
    }
}

void Deque::pushFront(int value) {
    /* Create the new cell to add. */
    Cell* cell = new Cell;
    cell->value = value;

    /* This cell is at the front of the list. */
    cell->next = head;
    cell->prev = NULL;

    /* If the list is empty, the new cell is now the sole element. */
    if (head == NULL) {
        head = tail = cell;
    }
    /* Otherwise, rewire the first element to point back at the new cell,
    * then update the head pointer.
    */
    else {
        head->prev = cell;
        head = cell;
    }
}
}
```

```

void Deque::pushBack(int value) {
    /* Create the new cell to add. */
    Cell* cell = new Cell;
    cell->value = value;

    /* This cell is at the back of the list. */
    cell->prev = tail;
    cell->next = NULL;

    /* If the list is empty, the new cell is now the sole element. */
    if (tail == NULL) {
        head = tail = cell;
    }
    /* Otherwise, rewire the last element to point into the new cell,
    * then update the tail pointer.
    */
    else {
        tail->next = cell;
        tail = cell;
    }
}

int Deque::popFront() {
    if (head == NULL) error("That which does not exist cannot be popped.");

    /* Cache the value to be removed, since we're going to free memory. */
    int result = head->value;
    Cell* toRemove = head;

    /* Advance the head to the next location. */
    head = head->next;

    /* There are two cases to consider. First, if the deque is nonempty,
    * then we need to break the backward-pointing link to the cell we're
    * about to remove.
    */
    if (head != NULL) {
        head->prev = NULL;
    }
    /* Otherwise, we have to also update the tail pointer to be NULL. */
    else {
        tail = NULL;
    }

    /* Reclaim memory. */
    delete toRemove;

    return result;
}

/* ... continued ... */

```

```

int Deque::popBack() {
    if (tail == NULL) error("That which does not exist cannot be popped.");

    /* Cache the value to be removed, since we're going to free memory. */
    int result = tail->value;
    Cell* toRemove = tail;

    /* Retreat the tail to the previous location. */
    tail = tail->prev;

    /* There are two cases to consider. First, if the deque is nonempty,
     * then we need to break the forward-pointing link to the cell we're
     * about to remove.
     */
    if (tail != NULL) {
        tail->next = NULL;
    }
    /* Otherwise, we have to also update the head pointer to be NULL. */
    else {
        head = NULL;
    }

    /* Reclaim memory. */
    delete toRemove;

    return result;
}

```

Problem Two: Quicksort Revisited

Here is one possible implementation:

```

/* Concatenates the two given lists together, updating first to
 * point to the new first cell of the concatenated list.
 */
void concatenateLists(Cell*& first, Cell* second) {
    /* Base case: If the first list is empty, we concatenate the
     * lists by just setting the second list to point to the first
     * list.
     */
    if (first == NULL) {
        first = second;
    }
    /* Recursive step: Otherwise, we concatenate the second list to
     * the list appearing after the first cell.
     */
    else {
        concatenateLists(first->next, second);
    }
}

/* Prepends the given single cell to the given list, updating the
 * pointer to the first element of that linked list.
 */
void prependCell(Cell* toPrepend, Cell*& list) {
    toPrepend->next = list;
    list = toPrepend;
}

```

/* ... continued on next page ... */

```

void partitionList(Cell* list, Cell*& smaller, Cell*& pivot, Cell*& bigger) {
    /* Distribute cells in the list into the three groups. */
    while (list != NULL) {
        /* Remember the next pointer, because we're going to remove this
         * element from the list it is currently in.
         */
        Cell* next = list->next;

        /* Determine which list this element belongs to. */
        if (list->value == pivot->value) {
            prependCell(list, pivot);
        } else if (list->value < pivot->value) {
            prependCell(list, smaller);
        } else {
            prependCell(list, bigger);
        }

        list = next;
    }
}

void quicksort(Cell*& list) {
    /* Determine the length of the list. If it's length 0 or 1, we're done. */
    if (list == NULL || list->next == NULL) return;

    /* Remove the first element as the pivot element. */
    Cell* pivot = list;
    Cell* rest = pivot->next;

    /* Remove the pivot element from the list. */
    pivot->next = NULL;

    /* Create two other lists: one of elements less than the pivot and one of
     * elements greater than the pivot.
     */
    Cell* smaller = NULL;
    Cell* bigger = NULL;

    /* Distribute the elements into the three lists based on
     * whether they are smaller than, equal to, or greater
     * than the pivot.
     */
    partitionList(rest, smaller, pivot, bigger);

    /* Recursively sort the two smaller regions. */
    quicksort(smaller);
    quicksort(bigger);

    /* Concatenate everything together. */
    concatenateLists(smaller, pivot);
    concatenateLists(smaller, bigger);

    /* The sorted list now begins with the element pointed at by the
     * smaller pointer.
     */
    list = smaller;
}

```